Voltage Overscaling Algorithms for Energy-Efficient Workflow Computations With Timing Errors

Aurélien Cavelan, Yves Robert, Hongyang Sun, Frédéric Vivien

Voltage Overscaling Algorithms for Energy-Efficient Workflow Computations With Timing Errors

Aurélien Cavelan[2,1], Yves Robert[1,2,3], Hongyang Sun[1,2] and Frédéric Vivien[2,1]
1. ENS Lyon, France
2. INRIA, France
3. University of Tennessee Knoxville, USA
aurelien.cavelan|yves.robert|hongyang.sun|frederic.vivien@inria.fr

ABSTRACT

We propose a software-based approach using dynamic voltage overscaling to reduce the energy consumption of HPC applications. This technique aggressively lowers the supply voltage below nominal voltage, which introduces timing errors, and we use Algorithm-Based Fault-Tolerance (ABFT) to provide fault tolerance for matrix operations. We introduce a formal model, and we design optimal polynomial-time solutions, to execute a linear chain of tasks. Evaluation results obtained for matrix multiplication demonstrate that our approach indeed leads to significant energy savings, compared to the standard algorithm that always operates at nominal voltage.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Fault tolerance

Keywords

Timing errors; ABFT; voltage overscaling; energy efficiency

1. INTRODUCTION

Reducing energy consumption has become a key challenge, for both economic and environmental reasons. In the scope of High-Performance Computing (HPC), *green computing* encompasses the design of energy-efficient algorithms, circuits, and systems. The dynamic power consumption of microprocessors is typically of the form $\alpha f V^2$, where α denotes the effective capacitance, f the frequency and V the operating voltage [1, 9]. One approach to reduce the energy consumption is thus to lower the frequency and/or the voltage at which cores operate. This approach is called *Dynamic Voltage and Frequency Scaling (DVFS)*. Lowering the supply voltage may seem the best option because it has a quadratic impact on the dynamic power, while frequency has only a linear impact. Voltage and frequency, however, cannot be set independently and at any value. Indeed, the lower the voltage, the higher the circuit latency, that is, the longer the delay for logic gates to produce their outputs. Therefore, for any frequency value, there is a minimal *threshold* or *nominal* voltage V_{TH}, at which the core can safely be used. In practice, given a choice for the frequency, one can always set the voltage at this threshold value to save energy, because using a higher voltage would lead to paying more energy without any benefit.

For a given frequency, if the core is used at a voltage below the nominal voltage V_{TH}, *timing errors* could occur, that is, the results of some logic gates could be used before their output signals reach their final values. The output of the circuitry could then be incorrect and the result of the overall computation, at the core level, could be incorrect. Here, we intentionally use the conditional "could" repeatedly. First, the whole circuitry can still produce a correct result even if some logic gates suffer from timing errors. Second, not all computation paths in a core have the same latency; the threshold voltage is computed for the worst case and not all computations correspond to the worst case. Third, there are process variations in the production of cores and, once again, the threshold voltage is defined so that the worst core works safely at that setting. For all these reasons, there is a significant probability that a computation performed below nominal voltage completes successfully, at least if the voltage is not "too low". Moreover, circuit manufacturers keep a safety margin. Cores are specified to be run under a supply voltage V_{DD}, which is considerably larger than the threshold ($V_{\text{DD}} > V_{\text{TH}}$). How to take advantage of these potential margins to reduce voltage and, thus, energy consumption?

A first approach is called *near-threshold computing* (NTC) [3, 10], where the supply voltage is chosen very close to (but larger than) the threshold voltage ($V_{\text{DD}} \approx V_{\text{TH}}$). Work in that scope mainly concerns the design of NTC circuits that operate safely and (almost) as quickly as non-NTC circuits, while providing great energy savings. A more aggressive approach is to use cores with a supply voltage *below* the threshold voltage ($V_{\text{DD}} < V_{\text{TH}}$), which is called *voltage overscaling* [6, 7]. Most existing work targeting voltage overscaling is hardware oriented and requires special hardware mechanisms to detect timing errors [6, 8, 7, 4]. Our work is among the very few existing purely software-based approaches that do not require any special hardware [11]. Because cores are operated below threshold voltage, they may be victims of timing errors, which may induce silent data corruptions (SDC): the output of the Arithmetic and Logic Unit (ALU) may be incorrect. Therefore, using cores in such a context requires to have mechanisms to detect these errors and to correct them. Furthermore, the energy cost of these detection and correction mechanisms should not offset the energy saving due to the low operational voltage.

One key characteristic of voltage overscaling makes it fundamentally different from most work on resilience for High-Performance Computing (HPC) applications. Indeed, a ubiq-

FTXS'15, June 15 2015, Portland, OR, USA

uitous assumption in HPC is that failures are random. In other words, like lightning, failures do not strike twice "at the same place". A consequence of this lightning assumption is used in most, if not all, fault-tolerant solutions for HPC. Assume that your computation has been the victim of a fault. Then, whatever your preferred fault-tolerant solution, you are going to re-execute your application one way or the other, in the same computational context. (This re-execution may be a temporal later-on re-execution, through a checkpoint-rollback mechanism, a simultaneous spatial re-execution through replication, etc.) Because of the lightning assumption, we know that with high probability the re-execution will not be the victim of the same failure. If we are unlucky, the re-execution will fail again, but because of another failure: the lightning will strike somewhere else.

On the contrary, faults in our context are timing errors: a signal is used before the processor circuitry has finished computing it. Therefore, these timing errors are *deterministic*: if the very same computation is performed in the very same context (temperature, voltage, value of operands, content of registers, history of instructions, etc.), the very same faulty result will be produced. In other words, the lightning always strikes twice! Consequently, none of the many existing solutions for dealing with failures in HPC can be used to cope with failures in a voltage overscaling context. Because timing errors are reproducible[1], we have no choice but to re-execute faulty computations in a different context.

In this paper, we investigate whether one can aggressively use voltage overscaling, in a purely software-based approach, to reduce the energy cost of executing a chain of tasks. We illustrate this approach using matrix multiplication. The bottom-line question is the following: is it possible to obtain the (correct) result of a matrix multiplication for a lower energy budget than that of the best DVFS solution? As in common blocking approaches such as [2], the matrix multiplication is decomposed into a series of multiplications of submatrices. In order to detect potential errors in the product of these submatrices, we use *Algorithm-Based Fault Tolerance (ABFT)* [5]. The idea is to execute an elementary matrix multiplication at a very low voltage and to check its correctness through ABFT. If the result is incorrect, we would then recompute it in another setting, that is, at a higher voltage, or in the worst case at nominal voltage. We assume the energy cost of executing an elementary matrix multiplication at each available voltage is known, as well as the probability of encountering timing errors at each voltage, through prior profiling. The algorithmic problem is then to decide, knowing these costs and probabilities, at which voltage to start executing the elementary matrix multiplications and at which voltage to re-execute them in case of failure. Should we go directly for the nominal voltage or should we risk once again an execution at a voltage below threshold?

We stress a major difference between our work and other algorithmic work focusing on finding "good" tradeoffs between performance (e.g., execution time, throughput) and energy consumption. In this work, we solely target energy minimization. This is because we use a fixed frequency (hence guaranteeing performance), and use aggressive voltage overscaling to save energy. The main contributions of this paper are:

- A formal model for the problem, which includes the mathematical consequences of the "failure strikes twice" property when computing conditional probabilities.
- An optimal polynomial-time strategy to execute either a single task or a chain of tasks.
- A set of numerical evaluations demonstrating that our approach does lead to significant energy savings.

The rest of this paper is organized as follows. In Section 2, we introduce a formal model for timing errors. We present the optimal algorithms in Section 3 and their evaluation in Section 4. We provide some final remarks in Section 5.

2. MODEL

In this section, we formally state our assumptions on timing errors. Then, we introduce the main notations. Finally, we investigate the impact of the assumptions on the conditional probabilities of success/failure. Because of the "lightning strikes twice" property, conditional properties are completely different from what is usually enforced for the resilience of HPC applications.

2.1 Timing errors

Silent errors caused by electro-magnetic radiation or cosmic rays strike the system randomly. On the contrary, timing errors are *deterministic* in nature. Suppose that a timing error has occurred under a given execution scenario (voltage, frequency, operand, etc.). Then the same error will occur with very high probability for another execution under the same scenario. Fundamentally, timing errors occur because one adjusts the system operating voltage below the threshold voltage V_{TH}, for a given frequency. Lowering the voltage increases the delay of the circuit, thereby potentially impacting the correctness of the computation. Different operations within the ALU may have different critical-path lengths. Similarly, for a given operation, different sets of operands may lead to different critical-path lengths (take a simple addition and think of a carry rippling to different gates depending upon the operands). In a nutshell, operations and operands are not equal with respect to timing errors.

In this paper, we focus on a fixed frequency environment (this frequency may have been chosen to achieve a given performance). Timing errors depend upon the voltage selected for execution, and we model this with the following two assumptions:

ASSUMPTION 1. *Given a computation and an input I, there exists a* threshold voltage $V_{\text{TH}}(I)$*: using any voltage V below the threshold ($V < V_{\text{TH}}(I)$) will always lead to an incorrect result, while using any voltage above that threshold ($V \geq V_{\text{TH}}(I)$) will always lead to a successful execution. Note that different inputs for the same computation may have different threshold voltages.*

ASSUMPTION 2. *When a computation is executed under a given voltage V, there is a probability p_V that the computation will fail, i.e., produces at least one error, on a random input. This failure probability is computed as $p_V = |\mathcal{I}_f(V)|/|\mathcal{I}|$, where $\mathcal{I}$ denotes the set of all possible inputs and $\mathcal{I}_f(V) \subseteq \mathcal{I}$ denotes the set of inputs for which the computation will fail at voltage V. Equivalently, $\mathcal{I}_f(V)$ is the set whose threshold voltage is strictly larger than V, according to Assumption 1.*

For any two voltages V_1 and V_2 with $V_1 \geq V_2$, we have $\mathcal{I}_f(V_1) \subseteq \mathcal{I}_f(V_2)$ (because of Assumption 1), hence $p_{V_1} \leq p_{V_2}$.

Since timing errors are essentially silent errors, they do not manifest themselves until the corrupted data has led to

[1] Although timing errors are deterministic, they cannot be forecast in practice. Indeed, this would require to consider all potential parameters in the execution: voltage, temperature, operands, content of registers, etc.

an unusual application behavior, which may be detected long after the error has occurred, wasting the entire computation done so far. Hence, an error-detection mechanism is necessary to ensure timely detection of timing errors, for instance, after the execution of each task. In this paper, we apply Algorithm-Based Fault Tolerance (ABFT), which uses checksums to detect errors. ABFT has been shown to work well on matrix operations with low overhead. However, we stress that the algorithms presented in Section 3 are fully general and agnostic of the error-detection technique (checksum, error correcting code, coherence tests, etc.).

To the best of our knowledge, the only paper targeting a pure algorithmic approach for near-threshold computing or voltage overscaling is [11]. The work in [11] also considers matrix multiplication using ABFT. However, it makes the classical assumption that failures do not strike twice, which does not apply to timing errors. Their approach only works when ABFT can detect and correct all the errors struck during the computation of an elementary matrix product. In practice, ABFT is limited to single error correction, which makes their approach viable only for small matrix blocks and infrequent errors. Timing errors may strike often when using very low voltages to decrease energy consumption.

2.2 Notations

We consider computational workflows that can be modeled as a chain of n tasks, $T_1, T_2, \ldots, T_n$, where T_{i+1} depends on T_i for $1 \leq i \leq n-1$. All tasks share the same computational weight, including the work to verify the correctness of the result at the end. Hence, all tasks have the same execution time and energy consumption under a fixed voltage and frequency setting. This framework applies to matrix multiplication, which we use to instantiate our model in Section 4. The system frequency is fixed (for application performance). To reduce energy consumption, we apply *dynamic voltage overscaling (DVOS)*, which enables a tradeoff between energy cost and failure probability. The platform can choose an operating voltage among a set $\mathcal{V} = \{V_1, V_2, \cdots, V_k\}$ of k discrete values, where $V_1 < V_2 < \cdots < V_k$. Each voltage V_ℓ has an *energy cost* per task c_ℓ that increases with the voltage, i.e., $c_1 < c_2 < \cdots < c_k$. Based on Assumption 2, each voltage V_ℓ also has a *failure probability* p_ℓ that decreases with the voltage, i.e., $p_1 > p_2 > \cdots > p_k$. We assume that the highest voltage V_k equals the *nominal* voltage V_{TH} with failure probability $p_k = 0$, thus guaranteeing error-free execution for all possible inputs. For convenience, we also use a *null* voltage V_0 with failure probability $p_0 = 1$ and null energy cost $c_0 = 0$.

Switching the operating voltage also incurs an energy cost. Let $o_{\ell,h}$ denote the energy consumed to switch the system operating voltage from V_ℓ to V_h. We have $o_{\ell,h} = 0$ if $\ell = h$ and $o_{\ell,h} > 0$ if $\ell \neq h$. Moreover, we assume that the voltage switching cost follows the *triangle inequality*, i.e., $o_{\ell,h} \leq o_{\ell,p} + o_{p,h}$ for any $1 \leq \ell, h, p \leq k$, which is true in practice. It basically means that, to switch from V_ℓ to V_h, no energy will be gained by first switching to an intermediate voltage V_p and then switching to the target voltage V_h.

The objective is to determine a sequence of voltages to execute each task in the chain, so as to minimize the expected total energy consumption.

2.3 Conditional probabilities

We now consider the implications of Assumptions 1 and 2 on the success and failure probabilities of executing a task following a sequence of voltages. For the ease of writing, we assume that the execution of each task has already failed under the null voltage V_0 (at energy cost $c_0 = 0$).

LEMMA 1. *Consider a sequence $\langle V_1, V_2, \cdots, V_m \rangle$ of m voltages, where $V_1 < V_2 < \cdots < V_m$, under which a given task is executed.*

(i) For any voltage V_ℓ, where $1 \leq \ell \leq m$, given that the execution of the task on a certain input has already failed under voltages $V_0, V_1, \cdots, V_{\ell-1}$, the probability that the task execution will fail under voltage V_ℓ on the same input is

$$\mathbb{P}(V_\ell\text{-fail} \mid V_0 V_1 \cdots V_{\ell-1}\text{-fail}) = \frac{p_\ell}{p_{\ell-1}}$$

(ii) For any voltage V_ℓ, where $1 \leq \ell \leq m$, let $\mathbb{P}(V_\ell\text{-fail})$ denote the probability that the task execution will fail under all voltages $V_0, V_1, \cdots, V_\ell$, and let $\mathbb{P}(V_\ell\text{-succ})$ denote the probability that the execution will fail under voltages $V_0, V_1, \cdots, V_{\ell-1}$ but will succeed under V_ℓ. We have

$$\begin{aligned} \mathbb{P}(V_\ell\text{-fail}) &= p_\ell \\ \mathbb{P}(V_\ell\text{-succ}) &= p_{\ell-1} - p_\ell \end{aligned}$$

PROOF. We prove property (i) using the fundamental assumptions on the error model — Assumptions 1 and 2. The task under study is the execution of some computation on some input I. Since this task execution has failed under voltages $V_0, V_1, \cdots, V_{\ell-1}$, we know that input I satisfies $I \in \bigcap_{h=0}^{\ell-1} \mathcal{I}_f(V_h) = \mathcal{I}_f(V_{\ell-1})$, where $\mathcal{I}_f(V_h)$ denotes the set of inputs on which the computation will fail under voltage V_h. Then, the task execution will fail under voltage V_ℓ if input I also falls in $\mathcal{I}_f(V_\ell) \subseteq \mathcal{I}_f(V_{\ell-1})$. Given that the input is randomly chosen (we have no a priori knowledge on it), the probability is $\mathbb{P}(V_\ell\text{-fail} \mid V_0 V_1 \cdots V_{\ell-1}\text{-fail}) = \frac{|\mathcal{I}_f(V_\ell)|}{|\mathcal{I}_f(V_{\ell-1})|} = \frac{|\mathcal{I}_f(V_\ell)|/|\mathcal{I}|}{|\mathcal{I}_f(V_{\ell-1})|/|\mathcal{I}|} = \frac{p_\ell}{p_{\ell-1}}$.

To prove (ii), we note that, in both cases, the task has failed under all voltages before V_ℓ. Using the result of (i), we get $\mathbb{P}(V_\ell\text{-fail}) = \prod_{h=1}^{\ell} \mathbb{P}(V_h\text{-fail} \mid V_0 \cdots V_{h-1}\text{-fail}) = \prod_{h=1}^{\ell} \frac{p_h}{p_{h-1}} = p_\ell$, and $\mathbb{P}(V_\ell\text{-succ}) = \left(\prod_{h=1}^{\ell-1} \mathbb{P}(V_h\text{-fail} \mid V_0 \cdots V_{h-1}\text{-fail})\right) \times \left(1 - \mathbb{P}(V_\ell\text{-fail} \mid V_0 \cdots V_{\ell-1}\text{-fail})\right) = \left(\prod_{h=1}^{\ell-1} \frac{p_h}{p_{h-1}}\right)\left(1 - \frac{p_\ell}{p_{\ell-1}}\right) = p_{\ell-1} - p_\ell$. □

3. OPTIMAL SOLUTION

In this section, we present a dynamic programming algorithm to minimize the expected energy consumption for executing a linear chain of tasks. We start with a single task (Section 3.1) before moving to a chain (Section 3.2).

3.1 For a single task

We first focus on a single task. The following lemma gives the expected energy consumption for any given voltage sequence starting at the current system voltage (preset voltage V_p before the first execution) and ending at the nominal voltage V_k (which guarantees successful completion).

LEMMA 2. *Suppose a sequence $L = \langle V_1, V_2, \cdots, V_m \rangle$ of m voltages is scheduled to execute a task, where $V_1 < V_2 < \cdots < V_m$, $V_1 = V_p$, and $V_m = V_k$. The expected energy consumption is*

$$E(L) = c_1 + \sum_{\ell=2}^{m} p_{\ell-1} \left(o_{\ell-1,\ell} + c_\ell\right) \tag{1}$$

PROOF. The task may be completed before all voltages in the sequence are used, so let $\mathbb{P}(V_\ell\text{-exec})$ be the probability that voltage V_ℓ is actually used to execute the task, which

happens when voltage $V_{\ell-1}$ has failed. Since the first voltage V_1 is always used, we have $\mathbb{P}(V_1\text{-exec}) = 1$, and the corresponding energy consumption is $\mathbb{P}(V_1\text{-exec})c_1 = c_1$. For $2 \le \ell \le m$, based on Lemma 1(ii), we have $\mathbb{P}(V_\ell\text{-exec}) = \mathbb{P}(V_{\ell-1}\text{-fail}) = p_{\ell-1}$, so the expected energy consumption of executing the task under V_ℓ is $\mathbb{P}(V_\ell\text{-exec})(o_{\ell-1,\ell} + c_\ell) = p_{\ell-1}(o_{\ell-1,\ell} + c_\ell)$. Summing up the expected energy of all voltages in the sequence leads to the result. □

Lemma 2 shows that the expected energy consumed by a voltage in any sequence is (only) related to the failure probability of the voltage immediately preceding it. We make use of this property to design a dynamic programming algorithm.

THEOREM 1. *To minimize the expected energy consumption for a single task, the optimal sequence of voltages to execute the task with any preset voltage $V_p \in \mathcal{V}$ can be obtained by dynamic programming with complexity $O(k^2)$.*

PROOF. Let L_s^* denote the optimal sequence of voltages among all possible sequences that start with voltage $V_s \in \mathcal{V}$, and when the system preset voltage is also at V_s. Let $E(L_s^*)$ denote the corresponding expected energy consumption by carrying out this sequence. According to Lemma 2, adding a new voltage before any sequence of voltages will only affect the expected energy of the first voltage in the original sequence. By using this property, we can formulate the following dynamic program to compute

$$\begin{aligned} E(L_s^*) &= c_s + \min_{s<\ell\le k} \{E(L_\ell^*) - c_\ell + p_s(o_{s,\ell} + c_\ell)\} \\ &= c_s + \min_{s<\ell\le k} \{E(L_\ell^*) + p_s o_{s,\ell} + (p_s - 1)c_\ell\} \quad (2) \end{aligned}$$

and the optimal sequence starting with voltage V_s is constructed as $L_s^* = \langle V_s, L_{\ell'}^*\rangle$, where

$$\ell' = \arg\min_{s<\ell\le k} \{E(L_\ell^*) + p_s o_{s,\ell} + (p_s - 1)c_\ell\}$$

The dynamic program is initialized with $E(L_k^*) = c_k$ and $L_k^* = \langle V_k\rangle$, and it is computed based on Equation (2) for $s = k-1, k-2, \cdots, 1$. The complexity is clearly $O(k^2)$. Note that the above formulation ensures every sequence L_s^*, for $s = 1, 2, \cdots, k$, ends with the nominal voltage V_k, so the task is guaranteed to be completed.

Given a preset voltage $V_p \in \mathcal{V}$, the optimal expected energy is then

$$E^*(V_p) = \min_{1\le s\le k} \{o_{p,s} + E(L_s^*)\}$$

and the optimal voltage sequence to execute the task with preset voltage V_p is

$$L^*(V_p) = L_{s'}^*$$

where $s' = \arg\min_{1\le s\le k} \{o_{p,s} + E(L_s^*)\}$. □

3.2 For a chain of tasks

We now present a dynamic programming algorithm to execute a linear chain $T_1 \prec T_2 \prec \cdots \prec T_n$ of n tasks. We point out that, due to the voltage switching cost, the optimal sequence of voltages to execute each task depends on the terminating voltage of the preceding task as well as the expected energy consumption to execute the subsequent tasks. Hence, the sequence could be very different for different task counts.

We first define some notations. Let $L_s^*(T_i)$ denote a sequence of voltages that starts with V_s for executing task T_i, and which leads to the optimal expected energy $E(\overrightarrow{T_i}, L_s^*(T_i))$ for executing the subchain $T_i \prec \cdots \prec T_n$. The optimal expected energy to execute $T_i \prec \cdots \prec T_n$ with any preset voltage $V_p \in \mathcal{V}$ is therefore $E^*(V_p, \overrightarrow{T_i}) = \min_{1\le s\le k}\{o_{p,s} + E(\overrightarrow{T_i}, L_s^*(T_i))\}$.

The following lemma gives the expected energy consumption to execute the subchain $T_i \prec \cdots \prec T_n$, given a voltage sequence for executing its first task T_i, the system preset voltage V_p (before the execution of T_i), and the optimal expected energy to execute the subsequent tasks.

LEMMA 3. *Suppose a sequence $L(T_i) = \langle V_1, V_2, \cdots, V_m\rangle$ of m voltages is scheduled to execute task T_i, where $V_1 < V_2 < \cdots < V_m$, $V_1 = V_p$, and $V_m = V_k$. The expected energy consumption to execute the subchain $T_i \prec \cdots \prec T_n$ by carrying out sequence $L(T_i)$ for task T_i and the optimal sequence for each subsequent task is*

$$\begin{aligned} &E(\overrightarrow{T_i}, L(T_i)) = c_1 + (1-p_1)E^*(V_1, \overrightarrow{T_{i+1}}) \\ &\quad + \textstyle\sum_{\ell=2}^{m} \left(p_{\ell-1}\,(o_{\ell-1,\ell} + c_\ell) + (p_{\ell-1} - p_\ell)\,E^*(V_\ell, \overrightarrow{T_{i+1}})\right) \end{aligned} \quad (3)$$

PROOF. As in Lemma 2, the expected energy consumed by carrying out sequence $L(T_i)$ for task T_i can be similarly computed to follow Equation (1), i.e., $c_1 + \sum_{\ell=2}^{m} p_{\ell-1}\,(o_{\ell-1,\ell} + c_\ell)$.

For a chain of tasks, the optimal sequence of voltages to execute each task depends on the terminating (successful) voltage of its preceding task. Suppose task T_i is successfully executed by voltage V_ℓ, then the optimal expected energy to execute the rest of the chain is $E^*(V_\ell, \overrightarrow{T_{i+1}})$. Based on Lemma 1(ii), for any $1 \le \ell \le m$, the probability that task T_i is successfully executed by V_ℓ is given by $\mathbb{P}(V_\ell\text{-succ}) = p_{\ell-1} - p_\ell$. Hence, the expected energy consumption by executing the remaining tasks is $\sum_{\ell=1}^{m} (p_{\ell-1} - p_\ell)\,E^*(V_\ell, \overrightarrow{T_{i+1}})$.

Summing up the expected energy for task T_i and for the rest of chain gives the result. □

THEOREM 2. *To minimize the expected energy consumption for a linear chain of tasks, the optimal sequence of voltages to execute each task, given the terminating voltage of its preceding task (or given the preset voltage $V_p \in \mathcal{V}$ for the first task), can be obtained by dynamic programming with complexity $O(nk^2)$.*

PROOF. Observe from Lemma 3 that the expected energy incurred by any voltage in a sequence to execute a task is related to the failure probability of the voltage itself as well as that of the immediately preceding voltage in the sequence. Hence, to determine the optimal voltage sequence to execute each task, we can establish the following dynamic program:

$$\begin{aligned} &E(\overrightarrow{T_i}, L_s^*(T_i)) \\ &\quad = c_s + (1-p_s)E^*(V_s, \overrightarrow{T_{i+1}}) + \min_{s<\ell\le k} \Big\{E(\overrightarrow{T_i}, L_\ell^*(T_i)) \\ &\qquad -c_\ell - (1-p_\ell)E^*(V_\ell, \overrightarrow{T_{i+1}}) + p_s\,(o_{s,\ell} + c_\ell) \\ &\qquad +(p_s - p_\ell)E^*(V_\ell, \overrightarrow{T_{i+1}})\Big\} \\ &\quad = c_s + (1-p_s)E^*(V_s, \overrightarrow{T_{i+1}}) + \min_{s<\ell\le k} \Big\{E(\overrightarrow{T_i}, L_\ell^*(T_i)) \\ &\qquad +p_s o_{s,\ell} + (p_s - 1)\left(c_\ell + E^*(V_\ell, \overrightarrow{T_{i+1}})\right)\Big\} \end{aligned}$$

for $1 \le i \le n-1$ and

$$\begin{aligned} &E(\overrightarrow{T_n}, L_s^*(T_n)) \\ &\quad = c_s + \min_{s<\ell\le k} \Big\{E(\overrightarrow{T_n}, L_\ell^*(T_n)) - c_\ell + p_s\,(o_{s,\ell} + c_\ell)\Big\} \\ &\quad = c_s + \min_{s<\ell\le k} \Big\{E(\overrightarrow{T_n}, L_\ell^*(T_n)) + p_s o_{s,\ell} + (p_s - 1)c_\ell\Big\} \end{aligned}$$

for $i = n$. The optimal voltage sequence $L_s^*(T_i)$ for each task T_i is constructed as $L_s^*(T_i) = \langle V_s, L_{l'}^*(T_i)\rangle$, where l' yields the minimum value of $E(\overrightarrow{T_i}, L_s^*(T_i))$ in the two equations above.

The dynamic program is initialized with $E(\overrightarrow{T_n}, L_k^*(T_n)) = c_k$ and $L_k^*(T_n) = \langle V_k\rangle$. For $1 \le i \le n-1$, it is initialized with $E(\overrightarrow{T_i}, L_k^*(T_i)) = c_k + E^*(V_k, \overrightarrow{T_{i+1}})$ and $L_k^*(T_i) = \langle V_k\rangle$. For each task T_i, starting from $i = n$, we first compute $E(\overrightarrow{T_i}, L_s^*(T_i))$ and construct $L_s^*(T_i)$ for all $s = k-1, k-2, \cdots, 1$. Then, for all $1 \le h \le k$, we need to compute the optimal expected energy to execute the subchain $T_i \prec \cdots \prec T_n$ when task T_{i-1} terminates at voltage V_h:

$$E^*(V_h, \overrightarrow{T_i}) = \min_{1\le s\le k}\{o_{h,s} + E(\overrightarrow{T_i}, L_s^*(T_i)\}$$

and the optimal voltage sequence to execute task T_i when task T_{i-1} terminates at voltage V_h:

$$L^*(V_h, T_i) = L_{s'}^*(T_i)$$

where $s' = \arg\min_{1\le s\le k}\{o_{h,s} + E(\overrightarrow{T_i}, L_s^*(T_i)\}$. After that, we can move on to task T_{i-1}. The optimal expected energy to execute the entire chain with preset voltage V_p is then given by $E^*(V_p, \overrightarrow{T_1})$, and we start executing the first task with voltage sequence $L^*(V_p, T_1)$. Again, the above formulation ensures the optimal sequence for each task ends with the nominal voltage V_k, so all tasks are guaranteed to be completed.

For the complexity, the computation of both $E(\overrightarrow{T_i}, L_s^*(T_i))$ and $E^*(V_h, \overrightarrow{T_i})$ for each task T_i takes $O(k^2)$ time, and it is clearly linear in the number of tasks. □

4. EVALUATION

In this section, we numerically evaluate the performance of the proposed dynamic programming solutions. We instantiate the application workflow with matrix multiplication and perform error detection (and correction) with ABFT.

4.1 Application workflow

Consider the blocked version of the inner-product algorithm for computing the matrix product $C = A \times B$, where A and B are square matrices of size $m \times m$. The block size is b, and the matrices are partitioned into $\lceil \frac{m}{b} \rceil^2$ blocks (or submatrices). Assuming that all elements of matrix C are initialized to zero, the following shows the sequential implementation of the algorithm:

for $i = 1$ to $\lceil \frac{m}{b} \rceil$ **do**
  for $j = 1$ to $\lceil \frac{m}{b} \rceil$ **do**
    for $k = 1$ to $\lceil \frac{m}{b} \rceil$ **do**
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} \times B_{k,j}$

which forms a chain of $n = \lceil \frac{m}{b} \rceil^3$ tasks with each task incurring $O(b^3)$ multiply-add operations. The block size b is chosen so as to enforce maximal cache re-use during the computation of one task. Setting b too small, however, incurs a larger overhead in loading and storing the data, thereby reducing the efficiency of the computation.

4.2 Algorithm-based fault tolerance

Algorithm-based fault tolerance (ABFT) is a technique developed by Huang and Abraham [5] to detect, locate and correct errors in matrix operations with low computational overhead. The idea is to add redundancy to the matrices in the form of checksums, which have been shown to be consistently maintained during the computation of many matrix operations. The following demonstrates the encoding scheme for the matrix multiplication $C = A \times B$.

First, define the *column checksum matrix* of matrix A as $A^c := \begin{pmatrix} A \\ e^T A \end{pmatrix}$, where $e = [1\ 1\ \cdots\ 1]^T$ is an all-one column vector. Define the *row checksum matrix* of matrix B as $B^r := \begin{pmatrix} B & Be \end{pmatrix}$. Finally, define the *full checksum matrix* of matrix C as $C^f := \begin{pmatrix} C & Ce \\ e^T C & e^T Ce \end{pmatrix}$. Instead of multiplying the original matrices A and B, we multiply the checksum matrices A^c and B^r, which produces the full checksum matrix C^f as follows:

$$\begin{aligned} A^c \times B^r &= \begin{pmatrix} A \\ e^T A \end{pmatrix} \times \begin{pmatrix} B & Be \end{pmatrix} \\ &= \begin{pmatrix} AB & ABe \\ e^T AB & e^T ABe \end{pmatrix} = \begin{pmatrix} C & Ce \\ e^T C & e^T Ce \end{pmatrix} = C^f \end{aligned}$$

Suppose an error has occurred during the above computation, then the checksum property in matrix C^f will no longer be satisfied, which can be easily detected by recomputing the checksums of C^f and comparing them to the results in the matrix product. Moreover, if only one error has occurred, then exactly one row and one column will violate the checksum property. In this case, we can locate the error (at the intersection of the inconsistent row and inconsistent column) and then correct it (by reversing the checksum computation). The same encoding scheme also works for matrix addition, thus we can apply it to detecting and (possibly correcting some) errors after each iteration of the blocked algorithm shown in the Section 4.1.

The overhead of performing ABFT on matrix blocks of size b, including the computation of the checksums themselves, the extra computation during the multiplication, and the error detection and correction, takes $O(b^2)$ operations, which is much lower than the $O(b^3)$ operations in the matrix multiplication for reasonable block sizes.

4.3 Evaluation setup

This section describes the various parameters used to instantiate the model. In the evaluation, we assume that timing errors occur only in the ALU, while the memory is protected and is thus error-free.

Matrix parameters.

We set the dimension of the matrices to be $m = 16384$, and vary the block size b in the evaluation. The maximum block size is set to be 256 for cache efficiency. The number of tasks is thus $n = \lceil m/b \rceil^3 = \lceil 16384/b \rceil^3$. For fault tolerance, the matrices are protected by the ABFT scheme described above. Hence, the computation of each task requires $w = b(b+1)^2 + \sigma$ operations, where σ denotes the overhead of initiating the matrix multiplication, which essentially prevents the use of very small blocks. In the evaluation, the overhead is assumed to be equivalent to multiplying two matrices of size 8×8, i.e., $\sigma = 8^3 = 512$. The time to compute each task is therefore $t = \tau \cdot w/\eta$, where $\tau = 1/f$ denotes the time to do one cycle at frequency f and η denotes the percentage of the peak processor performance that can be efficiently utilized. Since optimized matrix multiplication codes are known to be efficient, we set $\eta = 0.8$ in the evaluation.

Platform setting.

We adopt the set of voltages and the associated failure probabilities due to timing errors measured in [4] for a field-

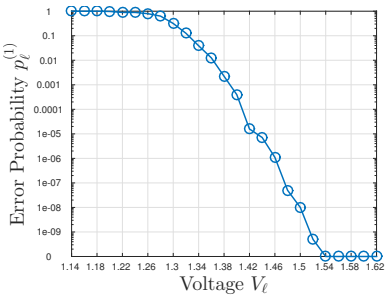


Figure 1: Set of voltages of an FPGA multiplier block and the associated error probabilities measured on random inputs at 90MHz and 27°C [4].

programmable gate array multiplier block at $f = 90$MHz and 27°C. Figure 1 shows the set $\mathcal{V}$ of available voltages, as well as the error probability $p_\ell^{(1)}$ of each voltage $V_\ell \in \mathcal{V}$ when performing a single operation on a random input. We take the *zero margin* voltage 1.54V that produces no error for all inputs as the nominal voltage V_k. As some errors can be corrected by hardware recovery mechanisms with little extra overhead, such as the technique reported in [4], they will not show up at the application level. Hence, we scale the associated error probability of each voltage by a factor of γ, which will be varied as a parameter in the evaluation. For any voltage $V_\ell \in \mathcal{V}$, the probability of having at least one error in the computation of a task with w operations can therefore be computed as $p_\ell = 1 - (1 - p_\ell^{(1)}/\gamma)^w$.

Energy cost.

The dynamic power consumption of microprocessors is typically modeled as $P(V, f) = \alpha f V^2$ [1, 9], where α denotes the effective capacitance, f the frequency, and V the operating voltage. By scaling the unit of power, we can assume wlog that $\alpha f = 1$ under a fixed frequency. Hence, for a given voltage V_ℓ and a block size b, the energy consumed to execute one task can be computed as $c_\ell = V_\ell^2 t$, where t is the time to execute the task. Note that we ignore the energy cost due to the $O(b^2)$ load and store operations in order to execute the task, which is incurred regardless of the execution algorithm.

The energy consumption to switch the operating voltage is assumed to be a linear function of the difference between the starting voltage and ending voltage. We model the switching energy as

$$o_{\ell,h} = \begin{cases} 0, & \text{if } \ell = h \\ \beta \cdot \frac{|V_\ell - V_h|}{V_k - V_1} & \text{otherwise} \end{cases} \tag{4}$$

where β denotes the cost to switch between the nominal voltage V_k and the lowest possible voltage V_1. In the evaluation, we will vary β to capture the relative cost of voltage switching compared to computing.

4.4 Evaluated algorithms

We evaluate the following algorithms and compare their performance. For all algorithms, the preset voltage is set to be the nominal voltage to begin the computation.

- *N-Voltage*: This is the baseline algorithm that applies near-threshold computing and always uses the nominal voltage $V_{\text{TH}} = V_k$ to execute all the tasks. Since many systems operate at a much higher supply voltage V_{DD} than the nominal voltage, their energy consumption is at least as high as the *N-Voltage* algorithm.
- *DP*$_1$*-detect* & *DP*$_1$*-correct*: These two algorithms use the dynamic program for a single task described in Section 3.1. Specifically, they apply the optimal sequence of voltages computed for one task to all the tasks in the chain. The expected energy consumption can be computed iteratively based on Equation (3). *DP*$_1$*-detect* uses ABFT for error detection, and in case of error it re-executes the task with a higher voltage. *DP*$_1$*-correct* also does error correction if exactly one error is detected, so that re-execution is not necessary in that case. If more than one error occurs, then *DP*$_1$*-correct* also re-executes the task.
- *DP*$_n$*-detect* & *DP*$_n$*-correct*: These two algorithms work similarly as the previous ones, using ABFT for error detection and correction, but they make direct use of the dynamic program for a chain of tasks described in Section 3.2. Hence, they are able to better take switching costs into account than *DP*$_1$*-detect* & *DP*$_1$*-correct*.

Due to the additional checksums in ABFT, the number of operations needed to execute each task in the DP-based algorithms is $b(b+1)^2$ instead of b^3. Because the *DP*$_n$*-correct* and *DP*$_1$*-correct* algorithms are able to correct up to one error, the failure probability of a voltage V_ℓ becomes the probability of having at least two errors in the execution, which for a task with w operations can be computed as

$$p_\ell = 1 - \left(1 - \frac{p_\ell^{(1)}}{\gamma}\right)^w - \binom{w}{1}\left(1 - \frac{p_\ell^{(1)}}{\gamma}\right)^{w-1} \frac{p_\ell^{(1)}}{\gamma}.$$

Since computational errors in matrix multiplications do not propagate, the above probability is a pessimistic estimation: indeed, more than two errors can be corrected if they happen to occur on the same element in matrix C. The extra overhead to correct an error is simply one additional operation.

4.5 Evaluation results

We now present the evaluation results. The first set of experiments is devoted to the evaluation of the algorithms when the voltage switching cost β is set to zero, and the probability scaling factor γ is fixed at 10.

Impact of block size b.

Figures 2(a) and 2(b) present the impact of block size b on the expected energy consumption. Figure 2(a) shows that using a small block size dramatically increases the energy consumption. This is partly due to the extra computation of ABFT, which reaches 13% $\left(= \frac{(16+1)^2 - 16^2}{16^2}\right)$ of the total computation when $b = 16$, and partly due to the extra overhead needed to handle the increasing number of blocks. For larger block sizes, the overhead becomes negligible and the energy consumption of the *N-Voltage* algorithm, which does not need ABFT, remains almost constant.

Surprisingly, the energy consumption of the *DP* algorithms does not seem to be much affected neither. In fact, decreasing the block size decreases the number of computations needed to execute one task, and thus the probability of failure. Then the algorithms can choose lower voltages, which saves energy, but the overhead of ABFT associated with smaller blocks increases. These gain and loss cancel out, so the energy consumptions of these algorithms remain quite stable as long as the block size b is not too small.

Figure 2(b) shows the normalized energy consumption of the algorithms with respect to the baseline algorithm *N-Voltage*. The *DP*$_n$*-correct* algorithm, which can tolerate one error, is less likely to fail than the *DP*$_n$*-detect* algorithm under the

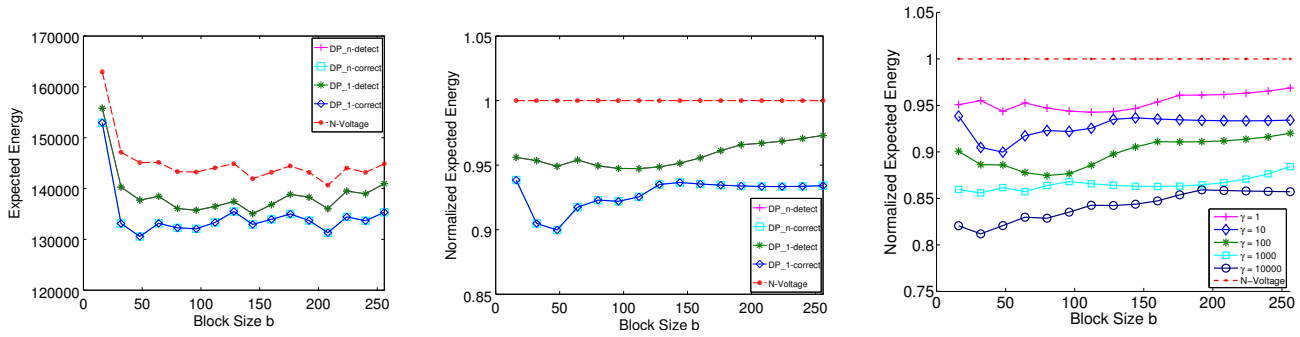


Figure 2: Impact of b and γ on the expected energy consumption for zero voltage switching cost. Only the results for the DP_n-*correct* algorithm are shown in (c).

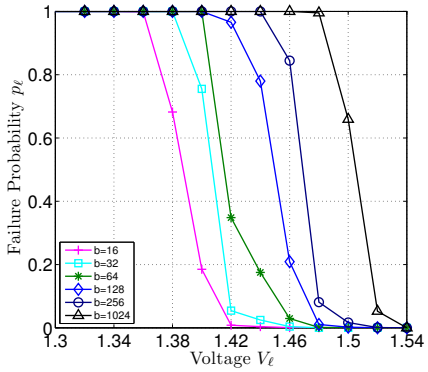


Figure 3: Failure probabilities for one task under different block sizes and voltages.

same voltage. This ability enables it to either choose a lower voltage while maintaining the same error probability, or use the same voltage while undergoing fewer failures. Both cases lead to savings in energy, between 5-10% in the experiment depending on the block size. Note that the DP_1 and DP_n algorithms yield the same energy consumption when the voltage switching cost is zero. In fact, the optimal sequence of voltages for one task turns out to be the same for all the tasks, which is the result of not having to pay the cost needed to reset the voltage after the completion of each task.

To better understand the performance of the algorithms, we plot in Figure 3 the failure probability of a single task under different voltages and block sizes. It shows that, for a given block size, there is at least one voltage below the nominal voltage with a failure probability that is low enough, so that the nominal voltage itself is almost never needed: the execution of a task will almost always succeed at a lower voltage. Therefore, the *DP* algorithms always yield better energy consumptions than the *N-Voltage* algorithm, as shown in Figure 2(b). This is only true for small block sizes (e.g., up to $b = 256$). For larger block sizes, such as $b = 1024$, only the nominal voltage can guarantee a failure-free execution.

Impact of probability scaling factor γ.

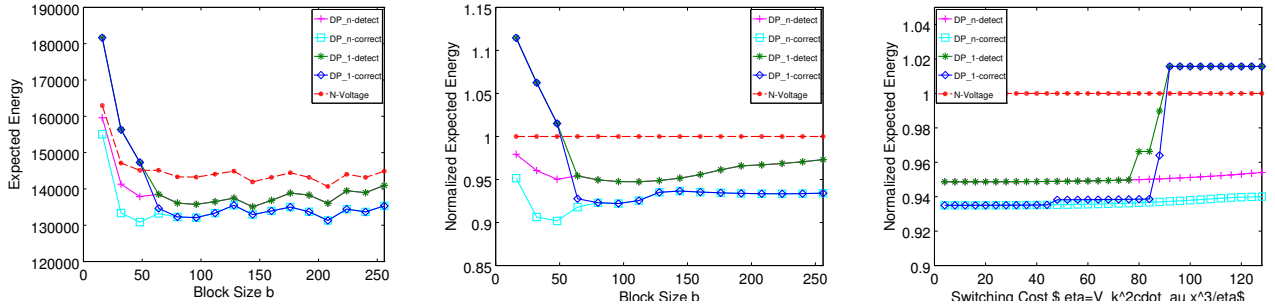
Figure 2(c) presents the effect of γ on the expected energy consumption of the DP_n-*correct* algorithm. Recall that γ is used to scale the error probabilities of the available voltages given in Figure 1 in order to account for the error-handling ability of the hardware. When γ equals 1, the probabilities are actually equal to the ones measured in [4]. Although it represents a pessimistic configuration, our algorithm still yields about 5% improvement over the *N-Voltage* algorithm. Higher values of γ offer more optimistic settings owing to better hardware error-handling technologies. This allows our algorithm to use lower voltages thus to save more energy. In particular, DP_n-*correct* is able to achieve nearly 15% energy saving compared to the baseline algorithm if there is a thousand-fold reduction in the failure probability of any single operation.

Impact of voltage switching cost β.

The second set of experiments focuses on the evaluation of the algorithms with non-negligible voltage switching costs. Figures 4(a) and 4(b) present the impact of voltage switching costs on the expected energy consumption of the algorithms under different block sizes. These figures are similar to Figures 2(a) and 2(b), except that the voltage switching cost between nominal and minimal voltages has been set to be equivalent to the energy consumed to multiply two 32×32 matrices at the nominal voltage without overhead, i.e., $\beta = V_k^2 \cdot \tau \cdot 32^3/\eta$. This is large enough to have an impact on the behavior of the algorithms.

Remember that the execution starts at the nominal voltage, so in order to lower the voltage for the first time, it is mandatory to pay some voltage switching cost. For the DP_1 algorithms, designed for a single task, it might not be beneficial to lower the voltage. This is especially true for small tasks, where the ratio between voltage switching cost and computation is relatively high. As a result, they tend to stick to the nominal (or a high) voltage, leading to more energy consumption. On the other hand, the DP_n algorithms consider the execution of the entire chain for deriving the optimal solution. In this case, the high voltage switching cost is amortized over all tasks. Hence, these algorithms will continue to explore the lower voltages, which, according to Figure 3, can still enjoy a good success probability for small tasks. Note that the *N-Voltage* algorithm never switches voltages and thus is not affected by the voltage switching cost.

Lastly, Figure 4(c) presents the impact of β when the block size is fixed to be $b = 128$. In particular, it shows the threshold values of β for which the DP_1 algorithms will stop exploring lower voltages and stick to the nominal voltage instead. By using the nominal voltage, they consume more energy and become worse than the DP_n algorithms, or even the *N-*



(a) (b) (c)

Figure 4: Impact of b and β on the expected energy consumption. In (a) and (b), the voltage switching cost is equivalent to the energy consumed to multiply two 32×32 matrices at nominal voltage without overhead, i.e., $\beta = V_k^2 \cdot \tau \cdot 32^3/\eta$.

Voltage algorithm because of the ABFT overhead. The result shows that for small voltage switching costs, both DP_1 and DP_n yield the same energy, but as soon as the switching cost reaches a threshold, only the more advanced DP_n algorithms are able to provide energy savings.

5. CONCLUSION

In this paper, we have proposed a software-based approach for reducing the energy consumption of HPC applications. The approach exploits dynamic voltage overscaling, which aggressively lowers the supply voltage below the nominal voltage. Because this technique introduces timing errors, we have used ABFT to provide fault tolerance for matrix operations. Based on a formal model of timing errors, we have derived an optimal polynomial-time solution to execute a linear chain of tasks. The evaluation results obtained for matrix multiplication demonstrate that our approach indeed leads to significant energy savings compared to the standard algorithm that always operates at (or above) the nominal voltage. The approach seems quite promising, and we plan to extend it to deal with other scientific application workflows.

Acknowledgment

This research was funded in part by the European project SCoRPiO, by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR), by the PIA ELCI project, and by the ANR RESCUE project. Yves Robert is with Institut Universitaire de France.

6. REFERENCES

[1] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[2] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5:173–184, 1996.

[3] R. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, 2010.

[4] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner. Razor: circuit-level correction of timing errors for low-power operation. *IEEE Micro*, 24(6):10–20, 2004.

[5] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, 1984.

[6] G. Karakonstantis and K. Roy. Voltage over-scaling: A cross-layer design perspective for energy efficient systems. In *European Conference on Circuit Theory and Design (ECCTD)*, pages 548–551, 2011.

[7] P. Krause and I. Polian. Adaptive voltage over-scaling for resilient applications. In *Proceedings of DATE*, pages 1–6, 2011.

[8] S. Ramasubramanian, S. Venkataramani, A. Parandhaman, and A. Raghunathan. Relax-and-retime: A methodology for energy-efficient recovery based design. In *Proceedings of DAC*, 2013.

[9] N. B. Rizvandi, A. Y. Zomaya, Y. C. Lee, A. J. Boloori, and J. Taheri. Multiple frequency selection in DVFS-enabled processors to minimize energy consumption. In A. Y. Zomaya and Y. C. Lee, editors, *Energy-Efficient Distributed Computing Systems*. John Wiley & Sons, Inc., 2012.

[10] M. Seok, G. Chen, S. Hanson, M. Wieckowski, D. Blaauw, and D. Sylvester. CAS-FEST 2010: Mitigating variability in near-threshold computing. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 1(1):42–49, 2011.

[11] T. M. Smith, E. S. Quintana-Orti, M. Smelyanskiy, and R. A. van de Geijn. Embedding fault-tolerance, exploiting approximate computing and retaining high performance in the matrix multiplication. In *Workshop On Approximate Computing (WAPCO)*, 2015.